Computer Networks Lab Assignment - I

Name: Arjeesh Palai

Class: BCSE UG - III

Section: A3

Roll Number: 002310501086

Problem Statement

Design and implement an error detection module which has two schemes namely Checksum and Cyclic Redundancy Check (CRC)

Design

Purpose of the Program

The project is a Windows TCP client–server that evaluates data-integrity using a 16-bit one's-complement checksum and multiple CRC variants (CRC-8, CRC-10, CRC-16 (IBM), CRC-32 IEEE). The client reads a 0/1 bitstream from a file, builds a codeword for the chosen scheme, and prepends a single-line metadata header– scheme, receiver_ip, receiver_port, client_ip, client_port, error_type, and data_len– before transmitting header+codeword to the server over TCP. On the wire, the checksum path pads the data to a 16-bit boundary, wraps carries during summation, and appends the non-zero complement; CRC codewords are formed by appending (g−1) zeros and the mod-2 division remainder using standard generator polynomials encoded as bit strings. To stress the detectors, the client can deliberately corrupt frames via four error models–single-bit, two isolated single-bit flips, an odd number of random flips, or a burst window–selected explicitly or at random. The server parses the header, extracts the bit payload, verifies it with the declared scheme (checksum sum == 0xFFFF or CRC remainder all zeros), and returns an application-level ACK string: ACCEPT (no error detected) or REJECT (error detected).

Structure Overview

- **Client Module (client.cpp):** Reads a 0/1 bitstream from msg.bits (filters out non-binary chars), selects a scheme (checksum16, crc8, crc10, crc16, crc32), builds the codeword (pads to 16-bit for checksum; appends CRC remainder for CRC), optionally corrupts it via the error injector, prepends an ASCII header (scheme, receiver_ip, receiver_port, client_ip, client_port, error_type, data_len) terminated by a newline, and sends header + codeword to the server over TCP (Winsock2).
- **Server Module (server.cpp):** Listens on a TCP port, accepts a connection, parses the ASCII header, extracts the bit payload, verifies integrity with the declared scheme (one's-complement 16-bit checksum sums to 0xFFFF, or CRC remainder is all zeros with the chosen generator), replies with a plain-text ACK: ACCEPT (no error detected) or REJECT (error detected), then handles clean disconnects.
- **Error Injection Module (error_injector.h):** Provides inject(bits, type) with four models to stress detection: single-bit flip, two isolated single-bit flips, an odd number of random flips, and a burst window. Can choose a specific type or a random one.
- **CRC & Checksum Module (common.h):** Supplies bit-string utilities and the encoding/verification logic: 16-bit one's-complement checksum (compute/verify) and CRC (encode/verify) using named generator polynomials mapped to schemes (crc8, crc10, crc16, crc32).

Input and Output Format

- Input
  - Data source: The client reads a path you pass on the CLI (arg 3) and loads a file of 0/1 characters (e.g., msg.bits); any non-binary characters are stripped before use.

    - Configuration: The error-detection scheme is selected via --scheme <checksum16|crc8|crc10|crc16|crc32>; the CRC generator polynomials are predefined in code and looked up by name.
    - Error injection controls: Enable injection with --inject yes|no, optionally set --inject-prob, choose a specific model with --injectscheme <0..3>, or let the program pick randomly; supported models are single-bit, two isolated single-bit flips, odd-count flips, and burst.
    - Transport & protocol: Client and server communicate over IPv4 TCP (Winsock2). The client prepends a one-line ASCII header—scheme;receiver_ip;receiver_port;client_ip;client_port;error_type;data_len—then sends header + codeword.
- Output
    - Server: Prints the parsed header, payload bit-length, client IP, and the verification result; it also returns a plaintext ACK: ACCEPT (no error detected) or REJECT (error detected).
    - Client: Logs bytes sent and the exact header it transmitted, then prints the server's ACK message when received.
    - Debug/error-injection traces: When enabled, the injector reports the selected model and flip locations (e.g., single-bit index or burst window and positions) in the console.

Implementation

SERVER MODULE (server.cpp)

• Architecture
- Single-threaded Winsock TCP server: WSAStartup → socket → bind → listen(backlog=1) → accept → recv → verify → send ACK → close.
- Each serve_once() handles exactly one client, then returns to accept for the next.

• Payload Format (over TCP)
- One ASCII header line, then '\n', then a bitstring codeword.
- Header is semicolon-delimited key=value fields:

  scheme;receiver_ip;receiver_port;client_ip;client_port;error_type;data_len

• Receive & Parse
- Read until a short recv; split buffer at first newline.
- Keep only '0'/'1' in the body (trim01). Parse header into a map.

• Verification
- If scheme == "checksum16": sum all 16-bit words (including appended checksum) with carry wrap; expect 0xFFFF.
- Else if scheme in {crc8, crc10, crc16, crc32}: mod-2 divide the received codeword by the named generator; remainder must be all zeros.

• Feedback
- Log header, body-length, client IP, and decision.
- Reply with plaintext: "ACCEPT (no error detected)" or "REJECT (error detected)".
- Close client socket; loop back to accept.

• Notes
- No Python files, no threads, no locks; connections are processed sequentially.

CLIENT MODULE (client.cpp)

• CLI & Setup
- Usage: client.exe <server_ip> <port> <input_bits_file>
  --scheme <checksum16|crc8|crc10|crc16|crc32>
  [--inject yes|no] [--inject-prob 0..1] [--injectscheme 0..3]
- Winsock client creates socket and connects to the given IPv4 endpoint.

• Data Loading
  - Read file, concatenate lines, and filter to retain only '0'/'1'. Exit if empty.

• Codeword Construction
  - checksum16: pad data to 16-bit boundary and append one's-complement checksum.
  - crc*: append (g−1) zeros, compute remainder with selected generator, append remainder.

• Optional Error Injection (before send)
  - Enable with --inject yes. Choose model via --injectscheme or pick randomly.
  - Models:
    SINGLE_BIT → flip one random bit
    TWO_ISOLATED → flip two separate bits (min gap enforced)
    ODD_ERRORS → flip 1/3/5 bits
    BURST → choose a window and flip several bits inside
  - Injector prints positions/summary to stderr for debugging.

• Wire Format & Send
  - Compose header fields exactly as:
    scheme=<...>;receiver_ip=<...>;receiver_port=<...>;

    client_ip=<...>;client_port=<...>;error_type=<...>;data_len=<N>\n
  - Send header + codeword in one write; wait for server ACK; print the ACK.

ERROR-DETECTION PRIMITIVES (common.h)

• Checksum-16 (one's complement)
  - Client: pad to 16 bits, compute checksum, append checksum to form codeword.
  - Server: re-sum all 16-bit words (including checksum) with carry wrap; accept if and only if total == 0xFFFF.

• CRC (bitstring arithmetic)
  - Generators available by name: "crc8", "crc10", "crc16" (IBM), "crc32" (IEEE).
  - Encoding: data + (g−1) zeros → mod-2 divide → append remainder.

    - Verification: mod-2 divide codeword; accept if and only if remainder is all zeros.

Code:

server.cpp:

```
#ifndef NOMINMAX
#define NOMINMAX
#endif
#define WIN32_LEAN_AND_MEAN

#include <iostream>
#include <string>
#include <sstream>
#include <unordered_map>
#include <cctype>
#include <cstdlib>
#include <winsock2.h>
#include <ws2tcpip.h>
#pragma comment(lib, "Ws2_32.lib")
#include "common.h"

using namespace std;

#define PAYLOAD_SIZE 64

class Server
{
    SOCKET listenfd{INVALID_SOCKET};
    sockaddr_in addr{};

public:
    Server(int port)
    {
        WSADATA wsa;
        if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) {
            cerr << "WSAStartup failed: " << WSAGetLastError() << "\n";
            exit(1);
        }

        listenfd = socket(AF_INET, SOCK_STREAM, 0);
        if (listenfd == INVALID_SOCKET) {
            cerr << "socket failed: " << WSAGetLastError() << "\n";
            exit(1);
        }

        int opt = 1;
        if (setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, (const
char*)&opt, sizeof(opt)) == SOCKET_ERROR) {
            cerr << "setsockopt failed: " << WSAGetLastError() << "\n";
```

```
            exit(1);
        }

        memset(&addr, 0, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_addr.s_addr = htonl(INADDR_ANY);
        addr.sin_port = htons((u_short)port);

        if (bind(listenfd, (sockaddr*)&addr, sizeof(addr)) == SOCKET_ERROR)
{
            cerr << "bind failed: " << WSAGetLastError() << "\n";
            exit(1);
        }
        if (listen(listenfd, 1) == SOCKET_ERROR) {
            cerr << "listen failed: " << WSAGetLastError() << "\n";
            exit(1);
        }
        cout << "[Server] Listening on port " << port << " ...\n";
    }

    ~Server()
    {
        if (listenfd != INVALID_SOCKET) closesocket(listenfd);
        WSACleanup();
    }

    void serve_once()
    {
        sockaddr_in cli{};
        int clen = sizeof(cli);
        SOCKET fd = accept(listenfd, (sockaddr*)&cli, &clen);
        if (fd == INVALID_SOCKET) {
            cerr << "accept failed: " << WSAGetLastError() << "\n";
            return;
        }
        cout << "[Server] Client connected.\n";

        string buf;
        char tmp[8 * PAYLOAD_SIZE];
        int n;
        while ((n = recv(fd, tmp, (int)sizeof(tmp), 0)) > 0) {
            buf.append(tmp, tmp + n);
            if (n < (int)sizeof(tmp)) break;
        }
        if (n == SOCKET_ERROR) {
            cerr << "recv failed: " << WSAGetLastError() << "\n";
        }

        size_t nl = buf.find('\n');
```

```
        if (nl == string::npos) {
            cerr << "Malformed payload (no header newline)\n";
            closesocket(fd);
            return;
        }
        string header = buf.substr(0, nl);
        string body = trim01(buf.substr(nl + 1));

        unordered_map<string, string> H;
        {
            stringstream ss(header);
            string kv;
            while (getline(ss, kv, ';')) {
                auto p = kv.find('=');
                if (p != string::npos) {
                    string k = kv.substr(0, p);
                    string v = kv.substr(p + 1);
                    auto trim = [](string& x) {
                        while (!x.empty() && isspace((unsigned
char)x.back())) x.pop_back();
                        size_t i = 0;
                        while (i < x.size() && isspace((unsigned
char)x[i])) ++i;
                        x = x.substr(i);
                    };
                    trim(k);
                    trim(v);
                    H[k] = v;
                }
            }
        }

        string scheme = H.count("scheme") ? H["scheme"] : "";
        string client_ip = H.count("client_ip") ? H["client_ip"] : "";
        string etype = H.count("error_type") ? H["error_type"] : "none";
        string data_len = H.count("data_len") ? H["data_len"] : "?";

        cout << "[Server] Header: " << header << "\n";
        cout << "[Server] Body bits length: " << body.size() << "\n";
        cout << "[Server] Client ip: " << client_ip << "\n";
        bool ok = false;
        if (scheme == "checksum16") {
            ok = checksum16_verify(body);
        } else if (is_crc_scheme(scheme)) {
            ok = crc_verify_codeword(body, crc_generators().at(scheme));
        } else {
            cerr << "[Server] Unknown scheme.\n";
        }
```

```
        cout << "[Server] Validation: " << (ok ? "ACCEPT (no error
detected)" : "REJECT (error detected)") << "\n";
        cout << "[Server] Meta:error_type=" << etype << "\n";
        string ack = (ok ? "ACCEPT (no error detected)" : "REJECT (error
detected)");
        send(fd, ack.c_str(), (int)ack.size(), 0);
        cout << "\nACK sent\n";
        closesocket(fd);
    }
};

static void usage()
{
    cerr << "Usage: Server.exe <port>\n";
}

int main(int argc, char** argv)
{
    if (argc != 2) {
        usage();
        return 1;
    }
    int port = stoi(argv[1]);
    Server r(port);
    while (true) r.serve_once();
    return 0;
}
```

client.cpp:

```
    #ifndef NOMINMAX
    #define NOMINMAX
    #endif
    #define WIN32_LEAN_AND_MEAN

    #include <iostream>
    #include <fstream>
    #include <sstream>
    #include <string>
    #include <algorithm>
    #include <cstdlib>
    #include <ctime>
    #include <cstdio>
    #include <cstring>

    #include <winsock2.h>
    #include <ws2tcpip.h>
    #pragma comment(lib, "Ws2_32.lib")

    #include "common.h"
    #include "error_injector.h"

    using namespace std;

    class Client {
```

```
    SOCKET sockfd{INVALID_SOCKET};
    sockaddr_in serv{};
    string ip;
    int port;

public:
    Client(const string& ip, int port) {
        WSADATA wsa;
        if (WSAStartup(MAKEWORD(2,2), &wsa) != 0) {
            cerr << "WSAStartup failed: " << WSAGetLastError() << "\n";
            exit(1);
        }

        this->ip = ip;
        this->port = port;

        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd == INVALID_SOCKET) {
            cerr << "socket failed: " << WSAGetLastError() << "\n";
            exit(1);
        }

        memset(&serv, 0, sizeof(serv));
        serv.sin_family = AF_INET;
        serv.sin_port   = htons((u_short)port);

        unsigned long a = inet_addr(ip.c_str());
        if (a == INADDR_NONE) {
            cerr << "invalid IPv4 address: " << ip << "\n";
            exit(1);
        }
        serv.sin_addr.s_addr = a;

        if (connect(sockfd, (sockaddr*)&serv, sizeof(serv)) ==
SOCKET_ERROR) {
            cerr << "connect failed: " << WSAGetLastError() << "\n";
            exit(1);
        }
    }

    ~Client() {
        if (sockfd != INVALID_SOCKET) closesocket(sockfd);
        WSACleanup();
    }

    static string read_bits_file(const string& path) {
        ifstream f(path);
        if (!f) {
            perror("open file");
            exit(1);
        }
        string all, line;
        while (getline(f, line)) all += line;
        return trim01(all);
    }

    static string make_codeword(const string& scheme, const string&
bits) {
        if (scheme == "checksum16") return checksum16_append(bits);
        if (is_crc_scheme(scheme))   return crc_make_codeword(bits,
crc_generators().at(scheme));
        cerr << "Unknown scheme: " << scheme << "\n";
        exit(1);
    }
```

```
    bool send_payload(const string& scheme, string& codeword,
                      bool injected, ErrorType etype, const string&
data_len_bits) {
        ostringstream hdr;

        sockaddr_in client_addr{};
        int len = sizeof(client_addr);
        if (getsockname(this->sockfd, (sockaddr*)&client_addr, &len) ==
SOCKET_ERROR) {
            cerr << "getsockname failed: " << WSAGetLastError() <<
"\n";
            return false;
        }

        char client_ip[INET_ADDRSTRLEN]{};
        const char* cip = inet_ntoa(client_addr.sin_addr);
        if (cip) {
            strncpy(client_ip, cip, INET_ADDRSTRLEN - 1);
            client_ip[INET_ADDRSTRLEN - 1] = '\0';
        } else {
            strcpy(client_ip, "0.0.0.0");
        }
        int client_port = ntohs(client_addr.sin_port);

        hdr << "scheme=" << scheme
            << ";receiver_ip=" << this->ip
            << ";receiver_port=" << this->port
            << ";client_ip=" << client_ip
            << ";client_port=" << to_string(client_port)
            << ";error_type=" << (injected ? errorTypeName(etype) :
"none")
            << ";data_len=" << data_len_bits
            << "\n";

        string header  = hdr.str();
        string payload = header + codeword;

        int n = send(sockfd, payload.c_str(), (int)payload.size(), 0);
        if (n == SOCKET_ERROR) {
            cerr << "send failed: " << WSAGetLastError() << "\n";
            exit(1);
        }
        cout << "[Client] Sent " << n << " bytes\n";
        cout << "[Client] Header: " << header;

        char buffer[1024] = {0};
        int valread = recv(sockfd, buffer, (int)sizeof(buffer), 0);
        if (valread > 0) {
            buffer[valread] = '\0';
            cout << "Received ACK from server: " << buffer << endl;
            string s(buffer);
            if (s.find("ACCEPT") != string::npos) return true;
        }
        return false;
    }
};

static void usage() {
    cerr << "Usage:\n"
            "  client.exe <server_ip> <port> <input_bits_file> --scheme
<checksum16|crc8|crc10|crc16|crc32>\n"
            "            [--inject yes|no] [--inject-prob 0..1]\n\n";
}

int main(int argc, char** argv) {
```

```
    string ip   = argv[1];
    int    port = stoi(argv[2]);
    string file = argv[3];

    string scheme;
    bool   inject = false;
    double inject_prob = 0.5;
    int    inject_scheme = -1;
    bool   random = false;

    for (int i = 4; i < argc; ++i) {
        string a = argv[i];
        if (a == "--scheme" && i + 1 < argc) scheme = argv[++i];
        if (a == "--inject" && i + 1 < argc) {
            string v = argv[++i];
            inject = (v == "yes" || v == "y" || v == "true" || v ==
"1");
        }
        if (a == "--inject-prob" && i + 1 < argc) {
            inject_prob = stod(argv[++i]);
            inject_prob = max(0.0, min(1.0, inject_prob));
        }
        if (a == "--injectscheme" && i + 1 < argc) inject_scheme =
stoi(argv[++i]);
        if (a == "--random" && i + 1 < argc) {
            string v = argv[++i];
            inject = (v == "yes" || v == "y" || v == "true" || v ==
"1");
        }
    }

    if (scheme.empty()) { usage(); return 1; }
    if (scheme != "checksum16" && !is_crc_scheme(scheme)) {
        cerr << "Invalid scheme\n"; return 1;
    }

    if (random) {
        srand((unsigned)time(nullptr));
        while (true) {
            string data_bits = Client::read_bits_file(file);
            if (data_bits.empty()) { cerr << "Input has no bits 0/1\n";
return 1; }
            string codeword = Client::make_codeword(scheme, data_bits);

            ErrorInjector inj;
            ErrorType etype = ErrorType::BURST;
            codeword = inj.inject(codeword, etype);

            Client s(ip, port);
            if (s.send_payload(scheme, codeword, true, etype,
to_string(data_bits.size()))) {
                cout << scheme << "\n" << codeword << "\n";
                break;
            }
        }
        return 0;
    }

    string data_bits = Client::read_bits_file(file);
    if (data_bits.empty()) { cerr << "Input has no bits 0/1\n"; return
1; }

    string codeword = Client::make_codeword(scheme, data_bits);
```

```
    ErrorInjector inj;
    bool actually_injected = false;
    ErrorType etype = static_cast<ErrorType>(inject_scheme);

    if (inject) {
        if (inject_scheme == -1) etype = inj.randomType();
        codeword = inj.inject(codeword, etype);
        actually_injected = true;
    }

    try {
        Client s(ip, port);
        s.send_payload(scheme, codeword, actually_injected, etype,
to_string(data_bits.size()));
    } catch (...) {
        cerr << "Client failed\n"; return 1;
    }
    return 0;
}
```

common.h:

```
#pragma once
#include <string>
#include <vector>
#include <unordered_map>
#include <cstdint>
#include <algorithm>

using std::string;
using std::unordered_map;
using std::vector;

inline string trim01(const string& s) {
    string t; t.reserve(s.size());
    for (char c : s) if (c == '0' || c == '1') t.push_back(c);
    return t;
}

inline string u16_to_bits(uint16_t x) {
    string b(16, '0');
    for (int i = 15; i >= 0; --i) { b[15 - i] = ((x >> i) & 1) ? '1' :
'0'; }
    return b;
}

inline uint16_t bits_to_u16(const string& b) {
    uint16_t v = 0;
    for (char c : b) { v = (uint16_t)((v << 1) | (c == '1')); }
    return v;
}

inline uint16_t checksum16_compute(const string& bits) {
    uint32_t sum = 0;
    for (size_t i = 0; i < bits.size(); i += 16) {
        uint16_t w = bits_to_u16(bits.substr(i, 16));
        sum += w;
        while (sum >> 16) sum = (sum & 0xFFFFu) + (sum >> 16);
    }
    uint16_t cs = (uint16_t)~sum;
    return cs == 0 ? 0xFFFFu : cs;
}
```

```
inline string checksum16_append(const string& data_bits) {
    string padded = data_bits;
    size_t rem = padded.size() % 16;
    if (rem) padded.append(16 - rem, '0');
    uint16_t cs = checksum16_compute(padded);
    return padded + u16_to_bits(cs);
}

inline bool checksum16_verify(const string& code_bits) {
    if (code_bits.empty() || (code_bits.size() % 16) != 0) return
false;
    uint32_t sum = 0;
    for (size_t i = 0; i < code_bits.size(); i += 16) {
        uint16_t w = bits_to_u16(code_bits.substr(i, 16));
        sum += w;
        while (sum >> 16) sum = (sum & 0xFFFFu) + (sum >> 16);
    }
    return (uint16_t)sum == 0xFFFFu;
}

inline string xor_block(const string& a, const string& b) {
    string r; r.resize(a.size());
    for (size_t i = 0; i < a.size(); ++i) r[i] = (a[i] == b[i]) ? '0' :
'1';
    return r;
}

inline string mod2_divide(const string& dividend, const string&
generator) {
    string rem = dividend;
    size_t g = generator.size();
    for (size_t i = 0; i + g <= rem.size(); ++i) {
        if (rem[i] == '1') {
            for (size_t j = 0; j < g; ++j) {
                rem[i + j] = (rem[i + j] == generator[j]) ? '0' : '1';
            }
        }
    }
    return rem.substr(rem.size() - (g - 1));
}

inline string crc_make_codeword(const string& data_bits, const string&
generator) {
    size_t rlen = generator.size() - 1;
    string padded = data_bits + string(rlen, '0');
    string rem = mod2_divide(padded, generator);
    return data_bits + rem;
}

inline bool crc_verify_codeword(const string& code_bits, const string&
generator) {
    string rem = mod2_divide(code_bits, generator);
    return std::all_of(rem.begin(), rem.end(), [](char c){ return c ==
'0'; });
}

inline const unordered_map<string, string>& crc_generators() {
    static const unordered_map<string, string> m = {
        {"crc8",  "100000111"},                  // x^8  + x^2 + x + 1
(CRC-8-ATM, with leading 1)
        {"crc10", "11000110011"},                // x^10 + x^9 + x^5 +
x^4 + x + 1
        {"crc16", "11000000000000101"},          // x^16 + x^15 + x^2 +
1 (CRC-16 (IBM))
        {"crc32", "100000100110000010001110110110111"} // CRC-32 (IEEE
```

```
802.3)
    };
    return m;
}

inline bool is_crc_scheme(const string& s) {
    const auto& m = crc_generators();
    return m.find(s) != m.end();
}
```

error_injector.h:

```
#pragma once

#ifndef NOMINMAX
#define NOMINMAX
#endif
#ifdef min
#undef min
#endif
#ifdef max
#undef max
#endif

#include <string>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <cmath>

enum class ErrorType {
    SINGLE_BIT = 0,
    TWO_ISOLATED = 1,
    ODD_ERRORS = 2,
    BURST = 3
};

inline const char* errorTypeName(ErrorType t) {
    switch (t) {
        case ErrorType::SINGLE_BIT:   return "single_bit";
        case ErrorType::TWO_ISOLATED: return
"two_isolated_single_bits";
        case ErrorType::ODD_ERRORS:   return "odd_number_of_errors";
        case ErrorType::BURST:        return "burst";
    }
    return "unknown";
}

class ErrorInjector {
public:
    ErrorInjector() { std::srand((unsigned)std::time(nullptr)); }

    std::string inject(const std::string& in, ErrorType type) {
        if (in.empty()) return in;
        std::string s = in;
        int n = (int)s.size();

        auto flip = [&](int pos) {
            s[pos] = (s[pos] == '0') ? '1' : '0';
        };
```

```
        switch (type) {
            case ErrorType::SINGLE_BIT: {
                int p = std::rand() % n;
                flip(p);
                std::cerr << "[Injector] SINGLE_BIT at " << p << "\n";
                break;
            }
            case ErrorType::TWO_ISOLATED: {
                if (n < 2) { flip(0); break; }
                int p1 = std::rand() % n;
                int p2 = std::rand() % n;
                while (p2 == p1 || std::abs(p2 - p1) < 2) p2 =
std::rand() % n;
                flip(p1);
                flip(p2);
                std::cerr << "[Injector] TWO_ISOLATED at " << p1 << ","
<< p2 << "\n";
                break;
            }
            case ErrorType::ODD_ERRORS: {
                int candidates[3] = {1, 3, 5};
                int flips = candidates[std::rand() % 3];
                std::cerr << "[Injector] ODD_ERRORS flips=" << flips <<
"\n";
                for (int i = 0; i < flips; ++i) flip(std::rand() % n);
                break;
            }
            case ErrorType::BURST: {
                if (n <= 3) { for (int i = 0; i < n; ++i) flip(i);
break; }

                int start = std::rand() % (n - 3);
                int len   = 3 + std::rand() % 32;
                int end   = std::min(start + len, n);

                std::cerr << "[Injector] BURST window start=" << start
                          << " len=" << (end - start) << "\n";

                int flips = 3 + std::rand() % (end - start);
                std::cerr << "[Injector] No of flips: " << flips;
                std::cerr << " [Injector] Flips Pos:- ";

                for (int i = 0; i < flips; ++i) {
                    int pos = start + std::rand() % (end - start);
                    flip(pos);
                    std::cerr << pos << " ";
                }
                std::cerr << "\n";
                break;
            }
        }
        return s;
    }

    ErrorType randomType() const {
        int v = std::rand() % 4;
        return static_cast<ErrorType>(v);
    }

    ErrorType chooseType(int a) const {
        return static_cast<ErrorType>(a);
    }
};
```

Test Cases

Test Case 1 — Single-Bit Error Detection

- Command (Checksum-16):
  client.exe 127.0.0.1 9000 msg.bits --scheme checksum16 --inject yes --injectscheme 0
- Command (CRC-16):
  client.exe 127.0.0.1 9000 msg.bits --scheme crc16 --inject yes --injectscheme 0
- Expected:
  Server → REJECT (error detected)
  Client → "Received ACK from server: REJECT (error detected)"
- Purpose:
  Validate both detectors catch any single flipped bit.

Test Case 2 — Two Isolated Single-Bit Errors

- Command (Checksum-16):
  client.exe 127.0.0.1 9000 msg.bits --scheme checksum16 --inject yes --injectscheme 1
- Command (CRC-16):
  client.exe 127.0.0.1 9000 msg.bits --scheme crc16 --inject yes --injectscheme 1
- Expected:
  CRC path → REJECT (error detected)
  Checksum-16 → usually REJECT, but MAY show ACCEPT for certain offset pairs (carry-wrap can mask some double-bit patterns).
- Purpose:
  Demonstrate CRC's advantage on many double-bit errors; show Checksum's potential miss.

Test Case 3 — Burst Error (contiguous window, random flips within)

- Command (Checksum-16):
  client.exe 127.0.0.1 9000 msg.bits --scheme checksum16 --inject yes --injectscheme 3
- Command (CRC-16) and (CRC-32):
  client.exe 127.0.0.1 9000 msg.bits --scheme crc16 --inject yes --injectscheme 3
  client.exe 127.0.0.1 9000 msg.bits --scheme crc32 --inject yes --injectscheme 3
- Expected:
  CRC-16 / CRC-32 → REJECT for typical short bursts (robust to burst errors up to their polynomial degree), CRC-32 being strongest.
  Checksum-16 → may ACCEPT depending on positions/length.
- Purpose:
  Assess robustness against burst-like corruption.

Test Case 4 — Odd Number of Errors (1/3/5 flips randomly placed)

- Command (Checksum-16):
  client.exe 127.0.0.1 9000 msg.bits --scheme checksum16 --inject yes --injectscheme 2
- Command (CRC-8/CRC-10/CRC-16/CRC-32):

client.exe 127.0.0.1 9000 msg.bits --scheme crc16 --inject yes --injectscheme 2

- Expected:
  CRC variants here generally detect odd-count error patterns; Checksum-16 may detect or miss depending on sums/carries.
- Purpose:
  Probe each scheme's behavior on non-structured multi-bit errors.

Test Case 5 — No Errors (clean transmission)

- Command (Checksum-16):
  client.exe 127.0.0.1 9000 msg.bits --scheme checksum16 --inject no
- Command (CRC-16):
  client.exe 127.0.0.1 9000 msg.bits --scheme crc16 --inject no
- Expected:
  Server → ACCEPT (no error detected)
  Client → "Received ACK from server: ACCEPT (no error detected)"
- Purpose:
  Confirm correct encode/verify paths without corruption.

Results and Discussion

Test Case	Sample Input Data	Error Introduced	Expected Output	Actual Observations
Single-Bit Error	1010101010101010	Flip a single bit (e.g., position 5) → **1010111010101010**	Checksum-16: **REJECT** •CRC-{8,10,16,32}: **REJECT**	Server ACK: **REJECT** for both schemes (single flips were always detected).
Two Isolated Single-Bit	1100110011001100	Flip two separate bits (e.g., 3 and 10) → **1110110011001110**	CRC-{16,32}: **REJECT**; CRC-{8,10}: usually **REJECT** •Checksum-16: may **ACCEPT**	CRC paths: **REJECT.** Checksum-16: mostly **REJECT,** but occasional **ACCEPT** for certain offset pairs.
Burst Error	1111000011110000	Burst window ≈4 starting at bit 4 → **1111011111110000** (multiple flips inside)	CRC-16/CRC-32: **REJECT** reliably •CRC-8/CRC-10: less robust on longer payloads • Checksum: may ACCEPT	CRC-16/CRC-32: consistently **REJECT;** CRC-8/CRC-10: occasionally weaker; Checksum-16: sometimes **ACCEPT.**
Odd Number of Errors	1011010110101101	Flip 3 bits (e.g., 2, 6, 13) → **1111010110111101**	CRC variants: generally **REJECT** •Checksum-	CRC variants: **REJECT** in runs tested.

Test Case	Sample Input Data	Error Introduced	Expected Output	Actual Observations
			16: may **ACCEPT** or **REJECT**	Checksum-16: mixed outcomes (pattern-dependent)
No Errors	0000111100001111	None	Checksum-16: **ACCEPT** •CRC-{8,10,16,32}: **ACCEPT**	Server ACK: **ACCEPT** for all schemes.

Console Evidence

• Server prints header fields (scheme, receiver/client endpoints, error_type, data_len), the payload bit length, and a decision line:
"ACCEPT (no error detected)" or "REJECT (error detected)".
• Client logs the exact header sent, bytes transmitted, and the ACK string received.

Summary of Findings

- **Error-Detection Accuracy:**
  - CRC showed consistently higher detection rates on complex patterns (two isolated flips, bursts, odd-count multi-bit), with CRC-32 and CRC-16 the most reliable.
  - Checksum-16 reliably caught single-bit flips, but could miss some even-count patterns (including certain two-bit and short burst cases) due to carry-wrap properties.

- **Performance:**
  - Both approaches are fast for the tested payload sizes. CRC with larger polynomials (CRC-32) incurs slightly more bit-operations than Checksum-16, but the overhead was negligible compared to socket I/O.

- **Correctness:**
  - Clean transmissions (no injection) were accepted by all schemes.
  - Error injections produced server REJECTs in line with theoretical guarantees for the chosen CRC polynomials and the observed limitations of one's-complement checksum.

- **Limitations:**
  - Short CRC polynomials (CRC-8/CRC-10) are less robust on longer payloads and certain burst geometries compared to CRC-16/CRC-32.
  - One's-complement checksum can yield false negatives for specific even-numbered error patterns that preserve the wrapped sum.

- **Potential Improvements:**
    - Prefer CRC-16 or CRC-32 for payloads beyond a few dozen bytes and for burst-heavy channels.
    - Consider adaptive selection of CRC polynomial based on payload length and required Hamming distance (e.g., default to CRC-32 for larger frames).